RMS\_SDK\_CPP Additions

## Design Inspiration

Several designs were explored for the Secure Sender System:

Seal Approach

The RMS SDK for C++ would provide a Seal object, which contained information about an email’s contents (sender, body, subject line, etc.). The Seal object also included a signed hash of the email’s contents, signed with the user’s CLC. When a client wished to send an email, they would provide the email object to the RMS SDK. It would then create a Seal and embed the seal in the email’s MIME headers. The client would then send the email normally.

When a client wishes to verify an email, they would provide the email to the SDK. The SDK would scan the email’s headers for a seal. If found, it would parse the seal out, and verify that the seal’s contents matched the signed hash attached to the seal. It would then check if the seal matched up with the email’s actual contents, and return a true/false value.

### DataCollection Approach

In this approach, the RMS SDK only provides two objects – a WritableDataCollection, and a DataCollection. The DataCollection is like a seal – a collection of data, with an attached signature signed with the user’s CLC. However, the DataCollection is not coupled with an email – it supports insertion of arbitrary key/value pairs. In addition, the DataCollection does not use the matching which a seal does. It simply verifies the signature against its own content, not against another object. A WritableDataCollection is like a DataCollection, but also supports inserting in key/value pairs

In this approach, a client has more to do. When sending an email, the client would create an WritableDataCollection, and insert any data about the email as key/value pairs into the WritableDataCollection. The client would use the WritableDataCollection’s SignAndSerialize() method to get a DataCollection JSON object, and attach this JSON to the email. When an email is received, the client scans the MIME headers for an embedded DataCollection. If found, it would generate a DataCollection object, and check if the DataCollection managed to self-verify (if it’s signature matches its contents). It would then manually check the email’s properties against the DataCollections, to verify authenticity.

### Design Chosen & Rational

The DataCollection approach was chosen. While the Seal approach leaves less for the client to do, the DataCollection approach is more versatile for the future. Down the road, if there is another use for a self-verifying collection of key-value pairs not related to email, a DataCollection could be used. A seal would not have this versatility, as it is closely attached to an email object. (Note: The seal model closely reflects what the Outlook Add-In for Secure Sender uses, but the Seal is implemented with a DataCollection base).

## DataCollection JSON Format

1. {
2. "data":{
3. “key”: “value” //All Key/Value pairs go here
4. },
5. "signature":{
6. "algorithm": string,
7. "certificate":{
8. "pld":{
9. "pub":{
10. "pld":{
11. "iss":{
12. "eurl": string,
13. "fname": string,
14. "id": string,
15. "iurl”: string,
16. "pubk":{
17. "e": string,
18. "kty": string,
19. "n": string
20. }
21. },
22. "issto":{
23. "em": string,
24. "id":{
25. "typ": string,
26. "val": string
27. },
28. "pubk":{
29. "e": string,
30. "kty": string,
31. "n": string
32. }
33. }
34. },
35. "sig":{
36. "alg": string,
37. "dig": string,
38. "penc": string
39. }
40. }
41. }
42. },
43. "signed\_data\_hash": string
44. }
45. }

rms\_sdk\_cpp\_wrapper

## Motivation

The RMS SDK for C++ is written in unmanaged C++, using Qt libraries. However, Outlook VSTO add-ins require a managed environment. While there are other ways to approach an Outlook add-in (such as an ATL add-in), these methods are poorly supported, and require an inordinate amount of work. To avoid this, the rms\_sdk\_cpp\_wrapper project has been created. This wraps several objects in the RMS SDK inside of C++ CLR objects, to allow them to be used in a managed environment. In addition, the rms\_sdk\_cpp\_wrapper\_tests project has been written in C#, to demonstrate usage of this project in a managed environment

## Important Notes

* **Reasoning for not using DLLImport:** DLLImport is designed to work with C, not C++. It does not preserve objects, only static functions. Since a large part of this SDK relies around various objects and their interactions, DLLImport would break a good chunk of the SDK’s functionality
* **IAuthenticationCallback Oddities:** IAuthenticationCallback is an interesting instance, as it is an interface within the RMS SDK for C++ which must be implemented by the client. However, it is not possible for a managed class to implement an unmanaged interface. As such, the workaround is as follows – an unmanaged object (IAuthenticationCallbackWrapper) has implemented the IAuthenticationCallback interface. In its constructor, IAuthenticationCallbackWrapper requires an IAuthenticationCallbackExp object, which is a managed interface with the same functions as a normal IAuthenticationCallback object. IAuthenticationCallbackWrapper calls IAuthentcationCallbackExp.GetAccessToken, unmarshalls the result, and returns it. Thus, managed clients can implement IAuthenticationCallbackExp. Every managed wrapper of an unmanaged class requiring an IAuthenticationCallback has been modified to create an IAuthenticationCallbackWrapper, fill it with the AuthenticationCallbackExp, and return it.

## Building Information

To build, first make sure that the proper directories have been marked under the “Additional Include Directories” option in Visual Studio. These directories are:

* (rms\_sdk\_cpp\_path)\sdk\rmsauth\_sdk\rmsauth
* (rms\_sdk\_cpp\_path)\sdk\rmscrypto\_sdk\CryptoAPI
* (rms\_sdk\_cpp\_path)\sdk\rms\_sdk
* (rms\_sdk\_cpp\_path)\third\_party\include
* (QT install directory)\5.7\msvc2015\include\QtCore
* (QT install directory)\5.7\msvc2015\include

In addition, the following libraries need to be marked as Additional Library Directories, under Linker settings:

* (rms\_sdk\_cpp\_path)\bin
* (QT install directory)\5.7\msvc2015\lib

Finally, these libraries need to be added under Linker->Input->Additional Dependencies

* Qt5Core.lib
* rms.lib

Outlook Add-In for Azure Secure Sender

# Overview

The Outlook Add-In for Azure Secure Sender is composed of three parts – The AzureSecureSender project, the SecureSenderLibrary project, and the SecureSenderLibraryTests project. The SecureSenderLibrary project contains all the code needed to attach signatures to emails, and verify those signatures. It also includes code to display a signature’s contents to a generic UI, given that the UI implements IUserInterface. The AzureSecureSender project contains the actual plug-in code, including code to call SecureSenderLibrary functions upon message send and receipt, as well as a UI that implements IUserInterface. SecureSenderLibraryTests includes unit tests for SecureSenderLibrary.

## SecureSenderLibrary

The SecureSenderLibrary is the bulk of the Azure Secure Sender plugin. It is completely abstracted from any Outlook code, through use of interfaces. The IEmail interface wraps an email, and any calling application must create their own implementation of IEmail to perform any email signing or verification. It also must implement an IUserInterface object to output data to the UI.

### Usage

Any client using this library must implement the IEmail interface. To perform email verification, call EmailHelper.GetDataIntegrityCheckForEmail on an IEmail object. A DataIntegrityCheck will be returned for the email. If the status is Verified, the IEmail is valid. If not, the status will reflect the issue with the email. To perform signing, call EmailHelper.SignEmail. The user’s email address must be provided, as well as an authentication callback (in case the RMS SDK for C++ needs to acquire a CLC).

To display to the UI, create a DataIntegrityCheckBinder object and implement the IUserInterface interface. When an email is verified, stage the resultant DataIntegrityCheck on the binder with ChangeData. Inflate whatever implementation of IUserInterface the client has implemented, and then call DisplayToUserInterface to output the data.

Important Objects

**IEmail:** An abstraction of an email object, used to separate the SecureSenderLibrary from any Outlook objects (such as MailItem, Outlook’s email object). Every “normal” property of the email (body, subject, sender), is represented as an EmailProperty object, which are stored in an EmailPropertiesList. This method of storing was used to easily allow for different IEmail implementations to have different numbers of properties, as well as variance within the property name themselves.

**IProperty:** Any property that can be inserted into a DataCollection. Has a Name and a Value. The interface can be implemented to add additional property values. For instance, the EmailProperty object implements this interface, and adds the additional “DisplayName” property to indicate which property should be shown on the UI.

**DataIntegrityCheck:** Parameterized to an implementation of IProperty. Checks an IEnumerable of IProperties against a DataCollection to find if they match. The DataIntegrityCheck holds the status of the check (failed server validation, doesn’t match, matches), as well as a list of properties which match the data collection, as well as a list of those which do not.

**IUserInterface:** An abstraction of a User Interface. Any implementation must support several key features, such as display error messages and tables.

**IUiDataBinder:** Acts as a shim between some data and a user interface. The data can be set using ChangeData, and BindToUserInterface displays the currently stored data on the user interface. This allows for data to be “staged” for display, and then displayed later. An implementation of this interface, DataIntegrityCheckBinder, outputs a DataIntegrityCheck to the user interface.

Important Notes

* **Header Folding:** It is recommended that MIME headers be no longer than 64 characters in length, although it appears they are cut off after 512 characters. Since a DataCollection could be longer than 64 characters, header folding is used. The JSON is subdivided into 64-character segments, and they are inserted into sequential headers (i.e. NAME-0, NAME-1, NAME-2, etc.). The header used is defined in the Resources file, and it is currently x-azure-secure-sender
* **Resources File:** All strings (such as error messages) are stored in the resources file, to make potential translation easier.

## Azure Secure Sender

Azure Secure Sender hooks the SecureSenderLibrary into an Outlook plugin. It implements IUserInterface as an Outlook adjoining form region, to display information about certificate statuses. This code is extremely simple – all the heavy lifting has been pushed off to the library.

### Important Notes

* **Adding/Removing DataCollection Properties:** The email properties which are verified are listed in EmailItem.cs, inside the EmailPropertiesList object. To add more properties, simply instantiate a new EmailProperty item with the desired name, value, and display name, and add it to this list. All the keys are stored inside the resources file
* **Outlook, External DLLs, and You:** Microsoft Outlook handles DLLs loaded by add-ins very oddly. Namely, the add-ins must be in a particular location – the calling DLL (Outlook)’s root directory. Hence, installing will require DLLs to be copied to Outlook’s root directory (as detailed below)

## Build Information

1. Ensure you have the same version of Outlook installed as your version of Qt on the machine. For example, if you are building Qt using msvc2015 (which is the name of the 32-bit compiler), make sure you have Office 32-bit installed.
2. Locate the install directory for Office (on my machine, it is C:\Program Files(x86)\Microsoft Office\root\Office16
3. Copy over all the files in the “bin” folder of your rms\_sdk\_cpp install over to this root folder.
4. Copy over every file from your Qt install into this folder
5. Click “Run” in Visual Studio